Creating Modular and Versatile Robotics Systems

Jonas Ferguson

Index Terms—ROS, Docker, robotics, computer vision, navigation, machine learning, containerization.

1 Introduction

This paper will detail the focus of my research during my enrollment in an Honors Research course over the Spring 2023 semester, working with the High-Performance Data-Intensive Computing Systems Laboratory at the University of Missouri. During the course of this semester, I continued to build on the research I did over the past year. The main focus of this research is building systems to enable future robotics-based research using Robot Operating System (ROS) and containerization with Docker [1] [2] [3].

1.1 Outline

I will first introduce Robot Operating System (ROS), detailing why it is used, how it works, and how I used it in my research. I will then discuss research I did in the field of computer vision. Finally, I will introduce containerization, Docker, and methods of combining the functionalities of ROS and Docker together.

2 Robot Operating System (ROS)

Contrary to the name, ROS is not an operating system, but instead an open-source project consisting of a set of tools and libraries to aid in robotics development [1].

2.0.1 Versions Used

While I used many versions of ROS in my research, I mainly used the ROS 2 distributions Foxy Fitzroy, run on Ubuntu Linux Focal Fossa (20.04), and Humble Hawksbill, run on Ubuntu Linux Jammy Jellyfish (22.04) [4] [5].

2.1 ROS Philosophy

ROS was created to be a peer-to-peer, open-source system for robotics that could be used to solve a wide variety of general robotics problems [1]. The intentionally general design of ROS makes it ideal for my goals, to create foundational systems that future robotics research can be based on.

2.1.1 ROS 1 vs. ROS 2

ROS (ROS 1) was originally released in 2010, and has since ceased development, with final support ending in 2025. ROS 2 was initially released in 2015 with support ongoing. While ROS 2 addresses a number of issues with ROS 1, for the purposes of my research, I was concerned with one main difference: ROS 1 uses a centralized architecture of nodes that communicate through a master node, whereas ROS 2 is developed as a decentralized collection of nodes that communicate as a network through DDS (Data Distribution Service) messages [5] [6]. This distinction is illustrated in [7, Fig.1]. This enables ROS 2 to function as a modular collection of nodes that can be swapped in and out as needed, as opposed to a centralized project structure, which is ideal for the creation of a general robotics research system. Additionally, the use of DDS messages allows ROS 2 to perform better and run on more systems, allowing it to be used in a wider variety of cases as well as supporting embedded systems [7]. For these reasons, my research focuses exclusively on the use of ROS 2, hereafter referred

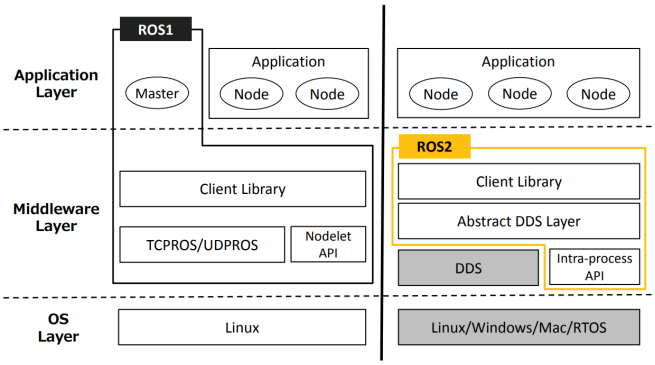

Fig. 1. ROS1 vs. ROS2 architecture.

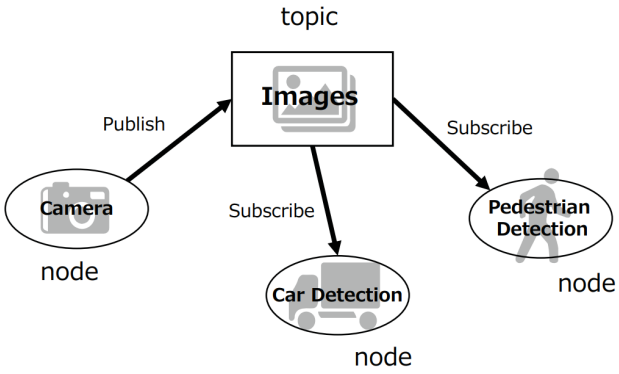

Fig. 2. Example of ROS nodes and topics.

to as ROS. Although, it is worth noting that ROS 1 and ROS 2 can be used together using the ROS bridge, which allows messages to be exchanged between the two versions. The wise ROS developer creates packages with a specific purpose that can then be used wherever they may be needed without needing to be modified to fit into a new project. This is the central idea in how I can create a multipurpose modular robotics research framework.

2.2 ROS Architecture

To understand the development of robotics systems using ROS, it is important to understand its underlying structure. ROS code is organized as packages that contain nodes and run inside a workspace. Nodes communicate using topics and can be launched with launch files.

2.2.1 ROS Workspaces

A ROS workspace is a directory that contains one or more ROS packages. It allows ROS packages to be built and sourced together and keeps ROS code organized. ROS distributions are installed as workspaces, which are sourced when using ROS to allow switching between distributions. Locally created workspaces run on top of the core ROS packages.

2.2.2 ROS Packages

All ROS code is organized in ROS packages, which are built within a workspace. A package ideally represents one general functionality and can then contain one or more nodes facilitating that functionality. Packages can be written in C++ and Python and can contain nodes, custom message types, service types, and build files that describe basic information about the package, what dependencies it needs, and how it should be built [8] [9]. The wise ROS developer uses the modular design of ROS packages to create packages with a specified purpose that can be moved around and used in many projects and use cases. Additionally, many open-source packages exist within the ROS community that can be combined with newly developed packages to avoid having to waste time recreating what already exists. It is crucial to note that package organization does not affect how ROS actually runs, that is done with nodes and the workspaces they are used in. Packages are used to organize code and allow it to be easily moved around.

2.2.3 ROS Nodes

ROS code is run using nodes, and a ROS system is composed of many nodes [1]. Like packages, nodes represent a single purpose. This often means creating packages with a single node, but it sometimes makes sense to have multiple nodes in a package if they are all needed to perform a task. Nodes communicate with each other (in ROS 2), and allow messages to move between the pieces in a system. Nodes can be run individually on the terminal, or this process can be automated, as seen with Docker in section 5.

2.2.4 ROS Topics

While topics are not the only way for nodes to communicate, they are the primary means of communication and the method which I was concerned with for my research. A ROS topic is defined as a name and a datatype which nodes can send data to and receive data from. Those sending data are called publishers, and they send messages under the given topic name and with the specified message type. Those receiving data are called subscribers, and they receive messages of the given message type under the given topic name. A single node can have many publishers and subscribers. This model allows for flexibility in communication — one-to-one, many-to-one, one-to-many, and many-to-many are all communication methods supported by ROS topics. ROS topics are asynchronous, which has worked well for my needs, but for projects requiring request-response communication, ROS offers a service-based architecture [5]. ROS topics can use pre-made message types defined for common use cases, but custom message types may also be defined, and can be used so long as the definition of the custom type is included wherever it is being used. An example of ROS nodes communicating over a ROS topic is illustrated in [7, Fig.2].

2.2.5 Launch Files

Nodes can be run manually, one at a time, but if a project will include many nodes being used together in a similar configuration, writing a launch file to do much of this work can save significant amounts of time. Section 5 will explore a solution with a similar principle using additional technology, whereas launch files are a part of the ROS architecture. Launch files can be written in Python, Extensible Markup Language (XML), or YAML Ain't Markup Language (YAML), and control the behavior of nodes, including when they start and stop and which parameters they are run with. The internal structure of launch files and exactly how they work varies between ROS 1 and ROS 2, but these details are not relevant to the scope of this paper, merely understanding the function of a launch file will suffice.

2.3 ROS Summary

With a basic understanding of the features of ROS, it is easy to see how it can be used in so many robotics systems to provide a wide-variety of functions. I will now detail some applications for ROS that I explored, although this list is by no means an exhaustive exploration of all ROS functionality.

3 Computer Vision

A field that I encountered several times during my research was that of computer vision. Of this, I was specifically using feeds of images from cameras, for the purposes of fiducial detection. This section will focus on the general infrastructure for making this work in ROS, as well as fiducial detection using AprilTags [10].

3.1 Image Pipeline

In order to get usable data from a camera to an AprilTag detection system, it needed to be moved around and processed, this is something that ROS can do very well. Using a modular approach to the design of packages and nodes that can be swapped around, my system uses a camera driver which varies by the camera being used, a time sync node, and an image rectifier using OpenCV [11].

3.1.1 Camera Driver

I used a variety of camera devices during my research and will give a brief rundown of how each camera was connected to the rest of the image pipeline.

3.1.1.1 Tello Drone: The Tello drone (Fig. 3), produced by DJI and Ryze Tech, is a very inexpensive device that can be easily used for basic robotics. It is a small quadcopter with a camera. I used it for my research because of its affordability, availability, and because it can be flown indoors. While I intend for this system to eventually be able to fly large drones that can collect vast amounts of data, this is impractical for work at this level, and the Tello drone makes a great substitute for this purpose. As a result of the widespread use of Tello drones in basic robotics, many libraries and packages for

Fig. 3. The DJI Tello drone

Fig. 4. The Intel RealSense camera.

connecting Tello drones to ROS exist. The Tello driver I used is written in Python, and adds additional functionality using the premade Tello SDK [12]. I was able to use this interface, which was built inside a ROS node, to communicate with the Tello over wi-fi, receive images from the drone, send controls to the drone, and also send these images out through ROS topics, and also receive new movement commands from ROS topics.

3.1.1.2 USB Webcam: While the Tello is a useful platform, for each use it has to be reconnected, requires battery charging, and adds significant complications simply for the fact that it moves. A simple wired USB webcam was something I had easy access to, and can be used with relatively few complications. Using USB Webcams with ROS is reasonably common, thus many libraries already exist for this purpose. The one I used is titled `usb\_cam` and is a library to connect USB cameras to ROS 1 or ROS 2 [13]. Using this library is quite simple, once a USB webcam is connected, the premade launch file is run, which then broadcasts out data. Data is broadcast to two ROS topics, `image\_raw` which sends images from the camera, and `camera\_info` which sends out data about the camera and calibration information.

3.1.1.3 RealSense Camera: While I did not use the RealSense camera for any practical purpose, I used it several times in setting up and testing systems that needed a camera, and it is thus worth mentioning here. Formally the Intel RealSense 435, it is a camera that supports not only the visible spectrum but also a sense of depth using infrared, shown in Fig. 4. ROS includes drivers that can communicate with the RealSense, and the data is communicated similarly to the USB webcam [14]. Additionally, NVIDIA has developed an entire ROS image pipeline using the RealSense, known as Isaac ROS which I used at times, but ultimately never used for any particular application [15].

3.1.2 Image Sync

The USB webcam and the RealSense camera both broadcast timestamped information to two separate topics regarding the images themselves (`image\_raw`) and camera information (`camera\_info`). The `usb\_cam` library in particular sends the `camera\_info` message only once, but the other drivers all send both topics together at every timestamp. This caused the timestamps to not be in sync, confusing later nodes along the pipeline. In order to fix this, I created a ROS package for my own use titled `image\_sync`. `image\_sync` functions very simply, it subscribes to the `image\_raw` and `camera\_info` topics. It saves the most recent `camera\_info` message whenever one is received. When an `image\_raw` message is

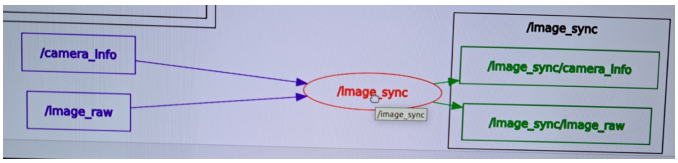

Fig. 5. The topics subscribed to and published by image_sync.

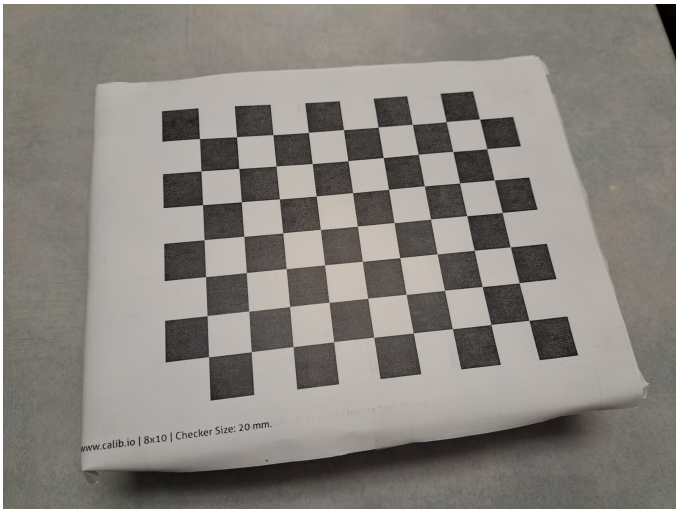

Fig. 6. A camera calibration checkerboard pattern.

received, it copies the timestamp from the `image\_raw` message onto the data of the most recent `camera\_sync` message. This leaves both topics with the same timestamp, and then they are published out. As the `image\_sync` and `camera\_info` topic names are used, I made use of namespaces within ROS so that the topic names append the name of the node broadcasting them. Thus, the published topics from `image\_sync` are `image\_sync/image\_raw` and `image\_sync/camera\_info`, as shown in Fig. 5.

3.1.3 Image Rectifier

In order to make use of images from a camera, that camera must be calibrated, and then the calibration data must be applied to each new frame to rectify the image and create usable data for robotic systems. Rectifying the image is important because it maps the image to an understanding of 3D space. This is how we can tell distance and direction in a 3D world from an image.

3.1.3.1 Camera Calibration: Camera calibration is fairly simple and needs to be done only once for a given camera. Camera calibration involves moving around a printed pattern in front of the camera so that data can be recorded about how shapes distort across the lens. While many patterns are available, I used a checkerboard pattern with 8 rows and 10 columns of 20 mm black and white squares (see Fig. 6). It is important to note that camera calibration software instead counts vertices where the squares connect, thus my calibration pattern would be considered 7 by 9. To use this calibration method, I used a premade ROS package called `camera\_calibration` [16]. The package has a node that is launched while specifying details about the size and shape of the chosen calibration pattern. From here, the pattern must be moved around the camera following four axes: X, Y, Size, and Skew. X and Y test how the camera distorts an image as it moves all across the lens. Size tests how the pattern is perceived as it is closer or farther from the camera. Finally, skew tests how the pattern looks as it is tilted. Once the node determines that all four axes have been sufficiently tested, the data is compiled into a standardized calibration file. The data from this file is then included in the `camera\_info` messages sent out by the camera driver in use. The camera calibration process is shown in Fig. 7.

3.1.3.2 Rectification: Once camera calibration data has been made, it can be used to rectify images. The computer vision library OpenCV simplifies this process greatly. OpenCV provides a function, `cv.undistort()`, which takes an image, and the information from camera calibration, and returns a rectified image [11].

3.2 AprilTags

AprilTag [10] is a visual feducial system which is common in the world of robotics, shown in Fig. 8. Visual feducials serve as landmarks to help robots locate themselves in an environment, so long as the feducial markers are kept in the same place relative to what they are marking. They can be thought of as similar

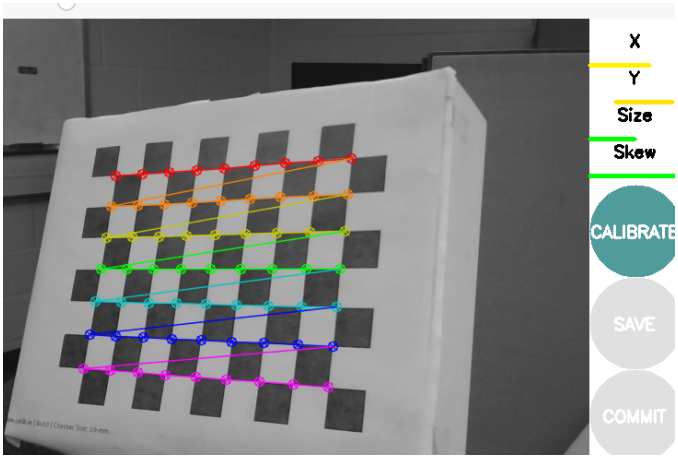

Fig. 7. The ROS camera calibration process.

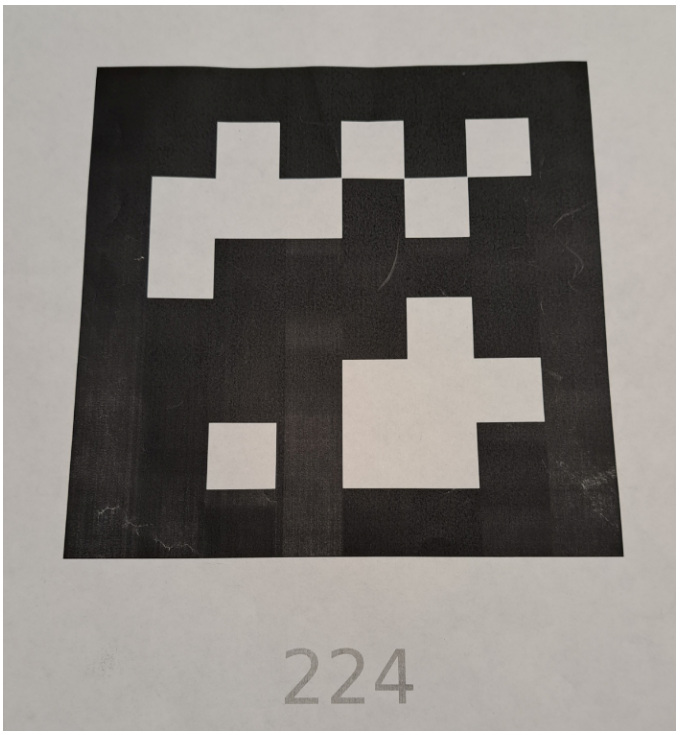

Fig. 8. An AprilTag with ID of 224.

to QR codes, although the intention differs. While many visual feducials exist, I chose to use AprilTags, as the project is open-source and much effort was put into creating a fast and reliable detector for the markers. Additionally, AprilTags are already in use at the College of Engineering, placed around Lafferre Hall for existing robots. Explaining the inner workings of AprilTags is beyond the scope of this paper, as premade sets of tags, as well as ROS wrappers of AprilTag detection algorithms, already

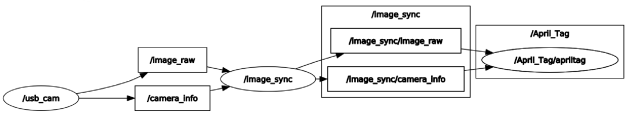

Fig. 9. AprilTag detection image pipeline.

exist.

3.2.1 *AprilTag Detection*

Many ROS implementations of AprilTags exist, after trying several versions, I ultimately used `apriltag\_ros` from Adlink-ROS [17]. This library is easy to run, as it is two ROS packages that are easily run with a premade launch file. It allows the detection of two sizes of AprilTag by default, but can be customized further. Additionally, it allows for all AprilTags to be detected or only specific AprilTags, depending on the needs of the project. Additionally, `apriltag\_ros` takes care of image rectification, provided the correct parameters are passed in through the `camera\_info` topic. Combining all these ideas together, it is possible to easily assemble a system that uses a camera driver to send images and camera information to `image\_sync` which syncs the two message topics. Which then sends them to `apriltag\_ros` which publishes to a new topic, `apriltag\_detections` whenever an AprilTag is detected. This message includes information on which AprilTag is detected and where that tag is in relation to the camera. A similar system to this is shown in Fig. 9. This last feature, information on where an AprilTag is located relative to the camera, will prove useful for localization and mapping, which will be discussed in the next section.

4 Navigation and Localization

4.1 ROS Transforms

ROS Transforms, present in the tf library in ROS, are a standard means of tracking locations and coordinates in a robotics system [18]. Robots often have many moving parts and need to keep track of where they all are in relation to each other. Attempting to re-implement this for

every robotics system is very time-consuming. In brief, tf uses pre-made files which represent a robotics system and then performs all the calculations necessary to find the location of any component of the system relative to any other. Recently, an update to this library, called tf2, has been released which expands on these ideas [19]. This is the version I used for my research.

4.2 Connecting AprilTags to ROS Transforms

AprilTags provide a convenient visual marker for using the tf library. The AprilTag detector, which I implemented in Section 3.2.1, is an example of a system that ROS transforms can be applied to. The detector uses a custom message type that provides a pose message along with the ID of the detected AprilTag. This pose message represents the location of the AprilTag relative to the camera. As the camera is not generally the center of the robot, and robots do not always stay stationary, we can use the tf library to instead find the distance from the center of the robot relative to the AprilTag, or relative to any other component in the system. From here, this information can allow us to navigate through space and determine where the robot is at any given point.

5 Deployment Solutions

When creating large and complex robotics projects, two problems appear repeatedly. One, it is time-consuming and tedious to have to constantly pull up various pieces of software, start them up, set parameters, etc., every time a test needs to be run. This problem only gets worse when considering how to ever deploy a solution that requires so much configuration. The second problem is that as projects grow, more pieces of software get compiled, each with their own dependency and version requirements. This ultimately leads to having to make difficult choices or complex workarounds in order to get software running together on one machine. Docker can be used to solve these problems by making an entire project run with one command, and having each piece of the project run in its own containerized environment where all resources and dependencies can be added without conflict.

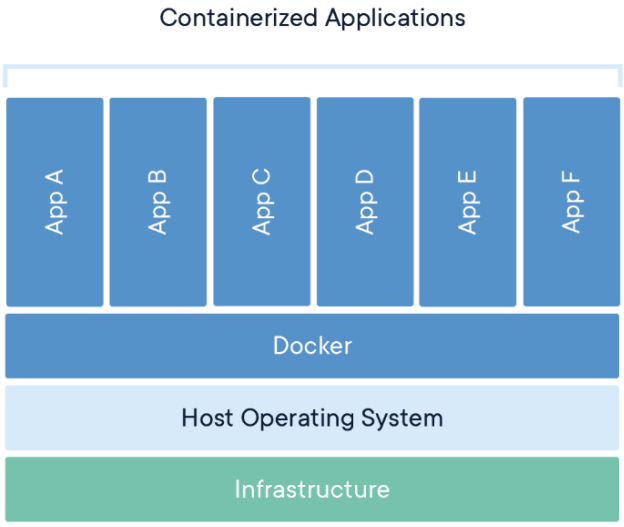

Fig. 10. Basic Docker container architecture.

5.1 Docker

Docker works by running within a container on the host system. It has many extensions of this concept that turn it into an entire ecosystem. The basic architecture of Docker containers is depicted in [3, Fig.10]. For my purposes, I am concerned only with using the Docker Engine, run through Docker Command Line Interface (CLI) and Docker Compose. Docker containers are run by the Docker Engine and function similarly to virtual machines, but they still run on the host operating system. Containers allow code to be run as independent processes on top of a host machine, with requiring less overhead than a virtual machine would. Docker containers are run from images, which can be downloaded or created, and represent a specific machine state that can be started from. Containers are interacted with through Docker CLI, which allows containers to be built, started, stopped, and many other features which are out of the scope of this paper.

5.1.1 Dockerfiles

Docker images can be pulled from Docker Hub and extended using custom Dockerfiles. Docker Hub is a repository for pre-made images, often operating systems or operating systems with specific pieces of software installed. For my purposes, I almost exclusively used Ubuntu images, and ROS installed on Ubuntu images. After pulling an image, further commands can be run inside the environment of that image using a Dockerfile to build custom images. One example of this was a Dockerfile that I wrote for using Flask, a Python library, to stream drone video to a web app [20].

```
FROM ubuntu:latest
```

All Dockerfiles start with the `FROM` instruction that says where they will pull a base image from. In this case, the base image is the latest installation of Ubuntu.

```
RUN apt-get update && \
        apt-get install -y \
    python3 \
    python3-pip \
    ffmpeg \
    libsm6 \
    libxext6
```

`RUN` commands such as this one will run the given text in the console of the container. In this case, we are updating the system's package repository and then installing Python and other dependencies to support it.

```
RUN pip3 install \
    flask \
    opencv-python \
    dnspython
```

Commands in a Dockerfile are always run sequentially, these packages are installed on top of Python to add additional functionality, and need to be installed after Python is.

```
COPY ./ ./
```

`COPY` commands can be run to copy any directory from the host machine into the specified directory path in the container. In this case, this will copy over the program for the Flash app itself.

```
EXPOSE 8080/udp
```

As Docker containers are separated from the computer's network by default, it is necessary to explicitly expose this port to the host machine so that the web app can be accessed outside the container.

```
CMD ["python3", "src/app.py"]
```

A Dockerfile may only have one `CMD` command. Unlike the other commands in a Dockerfile, the `CMD` command does not take effect while building the image. It instead serves as the default command when the image is run. Thus, once this image is built and is run, then the command `python3 src/app.py` will run in the container, starting up the Flask web app. Now that this Dockerfile has been created, we can build it by running `docker build ./path_to_image`. If it builds successfully, Docker will give an image ID, which we can use to run the container with `docker run container_id`. This process gets complicated when we have many containers, make adjustments to any, or begin to add more complex features and commands.

5.1.2 Docker Compose

Docker Compose allows for the routine building and running of containers following a Docker Compose file. While the same functionality exists within Docker CLI, Docker Compose vastly simplifies the entire process of building and running containers through a `docker-compose.yaml` file, which contains all the instructions for building and running many containers. I will now show an example Docker Compose file for running the Flask web app that I have just shown the Dockerfile for.

```
version: '3'
services:
```

All Docker Compose files begin by specifying the version. There are several elements that can follow, most notably is `services` which will specify exactly which Dockerfiles are being run and how they are to be configured.

```
  flask:
    container_name: flask_app
```

```
    build:
      dockerfile: Dockerfile.dev
      context: ./flash
```

`flask`: represents the start of a service, in this case, our Flask web app. We then specify a name for the container, the name of the Dockerfile, and then the file path of the directory containing the Dockerfile.

```
  volumes:
    - drone-camera:/src/static
```

This gives the container access to a Docker volume, which is a file space shared between containers. Specifically, access to the `drone-camera` Docker volume at the file path `/src/static`. A separate container places drone images in this volume, where Flask can find them and serve them into a web app. We will create this Docker volume later in the file.

```
  ports:
    - "5000:5000"
    - "1883:1883"
```

This allows us to map ports between the host machine and Docker container. Specifically, we are mapping ports 5000 and 1883 on the host machine to ports 5000 and 1883 in the container. It is also possible to change these port numbers as they are mapped, but this is unnecessary for this example.

```
  networks:
    - socket_network
```

This connects the Flask container to the Docker network `socket_network`. Docker networks allow containers to communicate to each other, the network itself is defined later in the file.

```
volumes:
  drone-camera:
```

This creates the `drone-camera` Docker volume. Note that this is not part of the `services` section, and is instead its own root-level section of the Docker Compose file.

```
networks:
  socket_network:
    driver: bridge
```

This creates the `socket_network` Docker network. This is also its own root-level section. The full version of this project includes the addition of two additional Docker containers, but sufficient example has been shown to understand the structure of the `docker-compose.yaml` file.

5.2 ROS in Docker

While Docker is very powerful by itself, by combining the functionality of ROS nodes with individual Docker containers, spun up using a Docker Compose file, we can create large projects with modular components that each run on their own dependencies and systems that can all be started with one command. This is the power of combining ROS and Docker functionality.

5.3 ROS Dockerfile

As ROS is a complex technology stack, working with it inside of Docker can get quite complicated. Using a ROS package involves three main steps, installing dependencies, building the package, and running the package. I will provide a basic example of a Dockerfile that can perform these functions in a Docker container. While this example can be heavily optimized, it serves as a good reference point for how Docker and ROS can be used together. This example is a Dockerfile to run the `image-sync` ROS node discussed in section 3.1.2.

```
FROM ros:humble
ARG OVERLAY_WS=/opt/ros/OVERLAY_WS
```

This sets up the rest of the file. We define the base image as ROS Humble, and set `OVERLAY_WS` to a specific file path in the container.

```
WORKDIR $OVERLAY_WS/src
COPY src/image_sync image_sync
```

Typically on the console, we move around directories using the `cd` command, and while this works with Dockerfiles, that new location does not persist past the given command. To move directories and stay there we instead specify the new directory with `WORKDIR`. Then, the ROS

files on the host system are copied over to the container so that we can build the ROS package.

```
WORKDIR $OVERLAY_WS
RUN apt-get update &&
        rosdep install -i \
    --from-paths \
        src \
    --ros-distro \
        humble \
    -y
```

This is how we can install dependencies within a Docker container. Before installing anything in a container, we must first run `apt-get update`, and then we can use the `rosdep` command as we would for building a ROS package locally.

```
RUN colcon build \
    --packages-select \
        image_sync
```

This functions effectively exactly as it does in ROS locally. We use the `colcon build` command to build the specified ROS package.

```
SHELL ["/bin/bash", "-c"]
RUN source install/setup.bash
```

In order to run the ROS package in a Docker container, we must use the `SHELL` command to enter the right environment. We can then source the newly built ROS package.

```
COPY ./ ./
```

Here, we copy over any additional files that are not specifically in the given ROS package file path. This is done at the end so that if a change is made to a file here, we do not have to rebuild the entire Dockerfile.

```
CMD ["ros2", "run",
    "image_sync", "image_sync_node"]
```

This final command starts up the ROS package, and will thus be run when we start up the image this Dockerfile builds. This command is the same one that would be used to run this package locally, just packaged into a Dockerfile. Many ROS nodes with similar Dockerfiles can then be packaged into a Docker Compose project and run all at once.

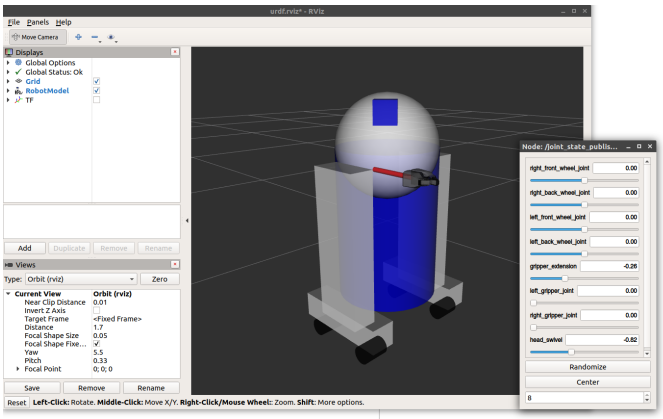

Fig. 11. An example visualization of a URDF file.

5.3.1 *Further Docker Examples*

While this section touched only on snippets of code, more complete samples of projects using ROS in Docker along with Docker Compose will be in the appendix of this paper.

6 Future Work

I have only scratched the surface of what can be done to create robotics systems. I will briefly explore a few possibilities for further work with what I have already done.

6.1 Pose Detection

For my research, I focused on using AprilTags as visual markers, but this does not have to be the case. With the use of algorithms to identify something else, the same concepts can be used to create systems with entirely different uses. One example is using a pose detection algorithm to allow robots to identify people in space, and to navigate with that knowledge in mind. An example scenario is to have a drone fly around a person and create a 3D representation of them. I did test out these concepts early on with a facial detection algorithm, but did not get far with using the data it created. This idea offers much further exploration.

6.2 Advanced Robots

When researching, I mostly stayed in the abstract, or to very basic robots. This system is

designed to work with robots of any complexity. Using a Unified Robot Description Format (URDF) file, any robot can be swapped into this system [21]. URDF files are used in ROS to represent all parts of a robot and their relation to one another. URDF files represent the physical shape of the robot, as well as what parts can move independently and how they can move [22, Fig.11]. Using this URDF file, the ROS robot state publisher can interpret the current state of the robot, and publish this information in a topic to the ROS transform library, which can then understand how all the components of the robot fit together in space [23]. Many commercially available robots have URDF files created already, which makes this a very helpful system to easily add different robots into.

7 Conclusion

While much work remains to be done, the work that I have done here provides a great foundation for the idea of creating general purpose robotics systems. The core of this project, the combination of ROS and Docker is beginning to become more wide-spread as more people realize its power, and as a result, this is becoming more common and in some ways easier. The principles explored here can be applied to any robotics system and problem.

Appendix
Further Examples

The files described in this section are in the folder attached to this paper and provide examples of Docker and ROS integration.

Drone Camera to Flask Web App Example

This example, partially shown in sections 5.1.1 and 5.1.2 is composed of three Docker containers: `flask`, `publisher`, and `subscriber`, all run with a Docker Compose file. The `publisher` container is responsible for connecting to a Tello drone, sending along its images, and controlling the movement of the drone. The `subscriber` container uses the images from the drone, applies a facial detection algorithm that highlights detected faces, and passes these images to the `flask` container. It also sends drone commands to the `publisher` container for following the detected face. The `flask` container takes the images from the `subscriber` and serves them to a simple web app. I have left only the basic file structure and Docker files, as that is the scope of what this paper is examining.

```
/
├── flask
│   ├── src
│   │   └── FLASK APP
│   └── Dockerfile.dev
├── publisher
│   ├── src
│   │   └── ROS PACKAGE
│   └── Dockerfile.dev
├── subscriber
│   ├── src
│   │   └── ROS PACKAGE
│   └── Dockerfile.dev
└── docker-compose.yaml
```

Apriltag Detector

This example, mentioned in section 3.2.1, is composed of three Docker containers, `webcam`, `image-sync`, and `apriltag-reader`, run with Docker Compose. The `webcam` container is responsible for connecting with a USB webcam, and passing along the webcam images as well as camera calibration information. The `image-sync` container takes this data from the webcam, and syncs the timestamps of the images to the camera information. Finally, the `apriltag-reader` container takes the synced images and camera information, determines if an Apriltag is present, and then publishes found Apriltags and their location relative to the camera to a ROS topic. As with the previous example, only the basic file structure and Docker files are shown.

```
/
└── apriltag-reader
    └── src
        ├── apriltag (ROS PACKAGE)
        └── apriltag_ros
            └── apriltag_msgs (ROS
                PACKAGE)
```

```
        └── apriltag_ros (ROS
            PACKAGE)
    ├── Dockerfile
├── image-sync
    ├── src
    │   └── image_sync (ROS PACKAGE)
    ├── Dockerfile
├── webcam
    ├── src
    │   └── usb_cam (ROS PACKAGE)
    ├── Dockerfile
└── docker-compose.yaml
```

ACKNOWLEDGMENTS

I would like to thank those who have helped me build this research from the beginning. Chris Scully most directly contributed to this work, helping to guide me in the right direction, teaching me concepts that I could not figure out on my own, and collaborating in building many of the examples seen in this paper. Ben Flink worked with me on parts of what is depicted here, as we worked together to both learn how to effectively use the Tello drone. Finally, I would like to thank Dr. Grant Scott for providing the knowledge, space, and resources for me to learn all the skills I have learned over the course of my research.

REFERENCES

[1] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng *et al.*, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.

[2] D. Thomas, W. Woodall, and E. Fernandez, "Next-generation ROS: Building on DDS," in *ROSCon Chicago 2014*. Mountain View, CA: Open Robotics, sep 2014. [Online]. Available: https://vimeo.com/106992622

[3] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[4] U. D. Project, *Official Ubuntu Documentation*, Canonical Ltd. [Online]. Available: https://help.ubuntu.com

[5] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot operating system 2: Design, architecture, and uses in the wild," *Science Robotics*, vol. 7, no. 66, p. eabm6074, 2022.

[6] G. Pardo-Castellote, "Omg data-distribution service: Architectural overview," in *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.* IEEE, 2003, pp. 200–206.

[7] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the performance of ros2," in *Proceedings of the 13th International Conference on Embedded Software*, ser. EMSOFT '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2968478.2968502

[8] ISO, *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, Feb. 2012. [Online]. Available: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372

[9] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009.

[10] E. Olson, "AprilTag: A robust and flexible visual fiducial system," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, May 2011, pp. 3400–3407.

[11] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[12] Damià Fuentes Escoté and Jakob Löw. (2023, January) Djitellopy. Commit e46c452. [Online]. Available: \url{https://www.github.com/damiafuentes/DJITelloPy}

[13] E. Flynn. (2023, April) usb_cam. Version 0.6.1. [Online]. Available: \url{https://www.index.ros.org/p/usb_cam/}

[14] D. Hirshberg, M. Curfman, S. Khshiboun, and B. Hirashima. (2022, September) realsense-ros. Release 4.51.1. [Online]. Available: \url{http://wiki.ros.org/RealSense}

[15] H. Shah, J. Singh, S. Wani, and H. Guillen. (2023, April) isaac_ros_image_pipeline. Release 0.30.0-dp. [Online]. Available: \url{https://github.com/NVIDIA-ISAAC-ROS/isaac_ros_image_pipeline}

[16] James Bowman and Patrick Mihelich and Vincent Rabaud and Joshua Whitley and Jacob Perron. (2022, June) camera_calibration. Version 3.0.0. [Online]. Available: \url{https://www.index.ros.org/p/camera_calibration/}

[17] H. Chen, T. Chang, and A. Synodinos. (2022, August) apriltag_ros ros2 node. Commit 2941821. [Online]. Available: \url{https://github.com/Adlink-ROS/apriltag_ros}

[18] T. Foote, "tf: The transform library," in *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*, ser. Open-Source Software workshop, April 2013, pp. 1–6.

[19] Tully Foote and Eitan Marder-Eppstein and Wim Meeussen. (2023, March) tf2. Version 0.25.2. [Online]. Available: \url{https://www.index.ros.org/p/tf2/}

[20] M. Grinberg, *Flask web development: developing web applications with python*. " O'Reilly Media, Inc.", 2018.

[21] Ioan Sucan and Jackie Kay and Chris Lalancette and Shane Loretz. (2022, March) urdf. Version 2.6.0. [Online]. Available: \url{https://www.index.ros.org/p/urdf/}

[22] Open Robotics. (2023) Building a movable robot model. [Online]. Available: \url{http://www.docs.ros.org/en/humble/Tutorials/Intermediate/URDF/Building-a-Movable-Robot-Model-with-URDF.html}

[23] Karsten Knese and Chris Lalancette and Shane Loretz. (2022, April) robot_state_publisher. Version 3.0.2. [Online]. Available: \url{https://www.index.ros.org/p/robot_state_publisher/}